下载APP

23｜函数：怎么让函数更简洁健壮？

2021-12-06 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 25:41 大小 23.52M

你好，我是 Tony Bai。

在上一节中，我们开始学习函数设计的相关知识，学习了如何基于 Go 函数的多返回值机制进行函数错误值构造方式的设计，你还记得那几种错误值构造与处理的策略吗？当然，良好的函数设计不仅仅要包含错误设计，函数的健壮性与简洁优雅也是我们在函数设计过程中要考虑的问题。

健壮的函数意味着，无论调用者如何使用你的函数，你的函数都能以合理的方式处理调用者的任何输入，并给调用者返回预设的、清晰的错误值。即便你的函数发生内部异常，函数也会尽力从异常中恢复，尽可能地不让异常蔓延到整个程序。而简洁优雅，则意味着函数的实现易读、易理解、更易维护，同时简洁也意味着统计意义上的更少的 bug。

这一节课，我们就将继续我们的函数设计之旅，聚焦在健壮与简洁这两方面，我们需要重点关注的内容。

我们先从健壮性开始。

健壮性的“三不要”原则

函数的健壮性设计包括很多方面，首先就有最基本的“三不要”原则，我们简单来分析一下。

原则一：不要相信任何外部输入的参数。

函数的使用者可能是任何人，这些人在使用函数之前可能都没有阅读过任何手册或文档，他们会向函数传入你意想不到的参数。因此，为了保证函数的健壮性，函数需要对所有输入的参数进行合法性的检查。一旦发现问题，立即终止函数的执行，返回预设的错误值。

原则二：不要忽略任何一个错误。

在我们的函数实现中，也会调用标准库或第三方包提供的函数或方法。对于这些调用，我们不能假定它一定会成功，我们一定要显式地检查这些调用返回的错误值。一旦发现错误，要及时终止函数执行，防止错误继续传播。

原则三：不要假定异常不会发生。

这里，我们先要确定一个认知：异常不是错误。错误是可预期的，也是经常会发生的，我们有对应的公开错误码和错误处理预案，但异常却是少见的、意料之外的。通常意义上的异常，指的是硬件异常、操作系统异常、语言运行时异常，还有更大可能是代码中潜在 bug 导致的异常，比如代码中出现了以 0 作为分母，或者是数组越界访问等情况。

虽然异常发生是“小众事件”，但是我们不能假定异常不会发生。所以，函数设计时，我们就需要根据函数的角色和使用场景，考虑是否要在函数内设置异常捕捉和恢复的环节。

在这三条健壮性设计原则中，做到前两条是相对容易的，也没有太多技巧可言。但对第三条异常的处理，很多初学者拿捏不好。所以在这里，我们就重点说一下 **Go 函数的异常处理设计**。

认识 Go 语言中的异常：panic

不同编程语言表示异常（Exception）这个概念的语法都不相同。在 Go 语言中，异常这个概念由 panic 表示。一些教程或文章会把它译为恐慌，我这里依旧选择不译，保留 panic 的原汁原味。

panic 指的是 Go 程序在运行时出现的一个异常情况。如果异常出现了，但没有被捕获并恢复，Go 程序的执行就会被终止，即便出现异常的位置不在主 Goroutine 中也会这样。

在 Go 中，panic 主要有两类来源，一类是来自 **Go 运行时**，另一类则是 **Go 开发人员通过 panic 函数主动触发的**。无论是哪种，一旦 panic 被触发，后续 Go 程序的执行过程都是一样的，这个过程被 Go 语言称为 **panicking**。

Go 官方文档以手工调用 panic 函数触发 panic 为例，对 panicking 这个过程进行了诠释：当函数 F 调用 panic 函数时，函数 F 的执行将停止。不过，函数 F 中已进行求值的 deferred 函数都会得到正常执行，执行完这些 deferred 函数后，函数 F 才会把控制权返还给其调用者。

对于函数 F 的调用者而言，函数 F 之后的行为就如同调用者调用的函数是 panic 一样，该 panicking过程将继续在栈上进行下去，直到当前 Goroutine 中的所有函数都返回为止，然后 Go 程序将崩溃退出。

我们用一个例子来更直观地解释一下 panicking 这个过程：

复制代码

```
1  func foo() {
2      println("call foo")
3      bar()
4      println("exit foo")
5  }
6
7  func bar() {
8      println("call bar")
9      panic("panic occurs in bar")
10     zoo()
11     println("exit bar")
12 }
13
14 func zoo() {
15     println("call zoo")
```

```
16      println("exit zoo")
17  }
18
19  func main() {
20      println("call main")
21      foo()
22      println("exit main")
23  }
```

上面这个例子中，从 Go 应用入口开始，函数的调用次序依次为`main -> foo -> bar -> zoo`。在 bar 函数中，我们调用 panic 函数手动触发了 panic。

我们执行这个程序的输出结果是这样的：

复制代码

```
1  call main
2  call foo
3  call bar
4  panic: panic occurs in bar
```

我们再根据前面对 panicking 过程的诠释，理解一下这个例子。

这里，程序从入口函数 main 开始依次调用了 foo、bar 函数，在 bar 函数中，代码在调用 zoo 函数之前调用了 panic 函数触发了异常。那示例的 panicking 过程就从这开始了。bar 函数调用 panic 函数之后，它自身的执行就此停止了，所以我们也没有看到代码继续进入 zoo 函数执行。并且，bar 函数没有捕捉这个 panic，这样这个 panic 就会沿着函数调用栈向上走，来到了 bar 函数的调用者 foo 函数中。

从 foo 函数的视角来看，这就好比将它对 bar 函数的调用，换成了对 panic 函数的调用一样。这样一来，foo 函数的执行也被停止了。由于 foo 函数也没有捕捉 panic，于是 panic 继续沿着函数调用栈向上走，来到了 foo 函数的调用者 main 函数中。

同理，从 main 函数的视角来看，这就好比将它对 foo 函数的调用，换成了对 panic 函数的调用一样。结果就是，main 函数的执行也被终止了，于是整个程序异常退出，日志"exit main"也没有得到输出的机会。

不过，Go 也提供了捕捉 panic 并恢复程序正常执行秩序的方法，我们可以通过 **recover 函数**来实现这一点。

我们继续用上面这个例子分析，在触发 panic 的 bar 函数中，对 panic 进行捕捉并恢复，我们直接来看恢复后，整个程序的执行情况是什么样的。这里，我们只列出了变更后的 bar 函数代码，其他函数代码并没有改变：

复制代码

```
func bar() {
    defer func() {
        if e := recover(); e != nil {
            fmt.Println("recover the panic:", e)
        }
    }()

    println("call bar")
    panic("panic occurs in bar")
    zoo()
    println("exit bar")
}
```

在更新版的 bar 函数中，我们在一个 defer 匿名函数中调用 recover 函数对 panic 进行了捕捉。recover 是 Go 内置的专门用于恢复 panic 的函数，它必须被放在一个 defer 函数中才能生效。如果 recover 捕捉到 panic，它就会返回以 panic 的具体内容为错误上下文信息的错误值。如果没有 panic 发生，那么 recover 将返回 nil。而且，如果 panic 被 recover 捕捉到，panic 引发的 panicking 过程就会停止。

关于 defer 函数的内容我们等会还会详细讲。此刻你只需要知道，无论 bar 函数正常执行结束，还是因 panic 异常终止，在那之前设置成功的 defer 函数都会得到执行就可以了。

我们执行更新后的程序，得到如下结果：

复制代码

```
call main
call foo
call bar
recover the panic: panic occurs in bar
exit foo
exit main
```

我们可以看到 main 函数终于得以“善终”。那这个过程中究竟发生了什么呢？

在更新后的代码中，当 bar 函数调用 panic 函数触发异常后，bar 函数的执行就会被中断。但这一次，在代码执行流回到 bar 函数调用者之前，bar 函数中的、在 panic 之前就已经被设置成功的 derfer 函数就会被执行。这个匿名函数会调用 recover 把刚刚触发的 panic 恢复，这样，panic 还没等沿着函数栈向上走，就被消除了。

所以，这个时候，从 foo 函数的视角来看，bar 函数与正常返回没有什么差别。foo 函数依旧继续向下执行，直至 main 函数成功返回。这样，这个程序的 panic“危机”就解除了。

面对有如此行为特点的 panic，我们应该如何应对呢？是不是在所有 Go 函数或方法中，我们都要用 defer 函数来捕捉和恢复 panic 呢？

如何应对 panic？

其实大可不必。

一来，这样做会徒增开发人员函数实现时的心智负担。二来，很多函数非常简单，根本不会出现 panic 情况，我们增加 panic 捕获和恢复，反倒会增加函数的复杂性。同时，defer 函数也不是“免费”的，也有带来性能开销（这个我们后面会讲解）。

那么，日常情况下我们应该怎么做呢？我这里提供了三点经验，你可以参考一下。

第一点：评估程序对 panic 的忍受度

首先，我们应该知道一个事实：**不同应用对异常引起的程序崩溃退出的忍受度是不一样的**。比如，一个单次运行于控制台窗口中的命令行交互类程序（CLI），和一个常驻内存的后端 HTTP 服务器程序，对异常崩溃的忍受度就是不同的。

前者即便因异常崩溃，对用户来说也仅仅是再重新运行一次而已。但后者一旦崩溃，就很可能导致整个网站停止服务。所以，**针对各种应用对 panic 忍受度的差异，我们采取的应对 panic 的策略也应该有不同**。像后端 HTTP 服务器程序这样的任务关键系统，我们就需要在特定位置捕捉并恢复 panic，以保证服务器整体的健壮度。在这方面，Go 标准库中的 http server 就是一个典型的代表。

Go 标准库提供的 http server 采用的是，每个客户端连接都使用一个单独的 Goroutine 进行处理的并发处理模型。也就是说，客户端一旦与 http server 连接成功，http server 就会为这个连接新创建一个 Goroutine，并在这 Goroutine 中执行对应连接（conn）的 serve 方法，来处理这条连接上的客户端请求。

前面提到了 panic 的“危害”时，我们说过，**无论在哪个 Goroutine 中发生未被恢复的 panic，整个程序都将崩溃退出**。所以，为了保证处理某一个客户端连接的 Goroutine 出现 panic 时，不影响到 http server 主 Goroutine 的运行，Go 标准库在 serve 方法中加入了对 panic 的捕捉与恢复，下面是 serve 方法的部分代码片段：

复制代码

```
// $GOROOT/src/net/http/server.go
// Serve a new connection.
func (c *conn) serve(ctx context.Context) {
    c.remoteAddr = c.rwc.RemoteAddr().String()
    ctx = context.WithValue(ctx, LocalAddrContextKey, c.rwc.LocalAddr())
    defer func() {
        if err := recover(); err != nil && err != ErrAbortHandler {
            const size = 64 << 10
            buf := make([]byte, size)
            buf = buf[:runtime.Stack(buf, false)]
            c.server.logf("http: panic serving %v: %v\n%s", c.remoteAddr, err,
        }
        if !c.hijacked() {
            c.close()
            c.setState(c.rwc, StateClosed, runHooks)
        }
    }()
    ... ...
}
```

你可以看到，serve 方法在一开始处就设置了 defer 函数，并在该函数中捕捉并恢复了可能出现的 panic。这样，即便处理某个客户端连接的 Goroutine 出现 panic，处理其他连接 Goroutine 以及 http server 自身都不会受到影响。

这种**局部不要影响整体**的异常处理策略，在很多并发程序中都有应用。并且，捕捉和恢复 panic 的位置通常都在子 Goroutine 的起始处，这样设置可以捕捉到后面代码中可能出现的所有 panic，就像 serve 方法中那样。

第二点：提示潜在 bug

有了对 panic 忍受度的评估，panic 是不是也没有那么“恐怖”了呢？而且，我们甚至可以借助 panic 来帮助我们快速找到潜在 bug。

C 语言中有个很好用的辅助函数，断言（assert 宏）。在使用 C 编写代码时，我们经常在一些代码执行路径上，使用断言来表达这段执行路径上某种条件一定为真的信心。断言为真，则程序处于正确运行状态，断言为否就是出现了意料之外的问题，而这个问题很可能就是一个潜在的 bug，这时我们可以借助断言信息快速定位到问题所在。

不过，Go 语言标准库中并没有提供断言之类的辅助函数，但我们可以使用 panic，部分模拟断言对潜在 bug 的提示功能。比如，下面就是标准库encoding/json包使用 panic 指示潜在 bug 的一个例子：

复制代码

```
// $GOROOT/src/encoding/json/decode.go
... ...
//当一些本不该发生的事情导致我们结束处理时，phasePanicMsg将被用作panic消息
//它可以指示JSON解码器中的bug，或者
//在解码器执行时还有其他代码正在修改数据切片。

const phasePanicMsg = "JSON decoder out of sync - data changing underfoot?"

func (d *decodeState) init(data []byte) *decodeState {
    d.data = data
    d.off = 0
    d.savedError = nil
    if d.errorContext != nil {
        d.errorContext.Struct = nil
        // Reuse the allocated space for the FieldStack slice.
        d.errorContext.FieldStack = d.errorContext.FieldStack[:0]
    }
    return d
}

func (d *decodeState) valueQuoted() interface{} {
    switch d.opcode {
    default:
        panic(phasePanicMsg)

    case scanBeginArray, scanBeginObject:
        d.skip()
        d.scanNext()

    case scanBeginLiteral:
        v := d.literalInterface()
        switch v.(type) {
```

```
33          case nil, string:
34              return v
35          }
36      }
37      return unquotedValue{}
38  }
```

我们看到，在valueQuoted这个方法中，如果程序执行流进入了 default 分支，那这个方法就会引发 panic，这个 panic 会提示开发人员：这里很可能是一个 bug。

同样，在 json 包的 encode.go 中也有使用 panic 做潜在 bug 提示的例子：

复制代码

```
1  // $GOROOT/src/encoding/json/encode.go
2
3  func (w *reflectWithString) resolve() error {
4      ... ...
5      switch w.k.Kind() {
6      case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32, reflect.Int6
7          w.ks = strconv.FormatInt(w.k.Int(), 10)
8          return nil
9      case reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32, reflect.
10         w.ks = strconv.FormatUint(w.k.Uint(), 10)
11         return nil
12     }
13     panic("unexpected map key type")
14 }
15
```

这段代码中，resolve方法的最后一行代码就相当于一个“代码逻辑不会走到这里”的断言。一旦触发“断言”，这很可能就是一个潜在 bug。

我们也看到，去掉这行代码并不会对resolve方法的逻辑造成任何影响，但真正出现问题时，开发人员就缺少了“断言”潜在 bug 提醒的辅助支持了。在 Go 标准库中，**大多数 panic 的使用都是充当类似断言的作用的。**

第三点：不要混淆异常与错误

在日常编码中，我经常会看到一些 Go 语言初学者，尤其是一些有过 Java 语言编程经验的程序员，会因为习惯了 Java 那种基于try-catch-finally的错误处理思维，而将 Go

panic 当成 Java 的“checked exception”去用，这显然是混淆了 Go 中的异常与错误，这是 Go 错误处理的一种反模式。

查看 Java 标准类库，我们可以看到一些 Java 已预定义好的checked exception类，比较常见的有IOException、TimeoutException、EOFException、FileNotFoundException，等等。看到这里，你是不是觉得这些 checked exception 和我们上一节讲的“哨兵错误值”十分相似呢？。它们都是预定义好的、代表特定场景下的错误状态。

那 Java 的checked exception 和 Go 中的 panic 有啥差别呢？

Java 的checked exception用于一些可预见的、常会发生的错误场景，比如，针对checked exception的所谓“异常处理”，就是针对这些场景的**“错误处理预案”**。也可以说对checked exception的使用、捕获、自定义等行为都是“**有意而为之**”的。

如果它非要和 Go 中的某种语法对应来看，它对应的也是 Go 的错误处理，也就是基于 error 值比较模型的错误处理。所以，Java 中对checked exception处理的本质是**错误处理**，虽然它的名字用了带有“异常”的字样。

而 Go 中的 panic 呢，更接近于 Java 的RuntimeException+Error，而不是checked exception。我们前面提到过 Java 的checked exception是必须要被上层代码处理的，也就是要么捕获处理，要么重新抛给更上层。但是在 Go 中，我们通常会导入大量第三方包，而对于这些第三方包 API 中是否会引发panic，我们是不知道的。

因此上层代码，也就是 API 调用者根本不会去逐一了解 API 是否会引发panic，也没有义务去处理引发的 panic。一旦你在编写的 API 中，像checked exception那样使用 panic 作为正常错误处理的手段，把引发的panic当作错误，那么你就会给你的 API 使用者带去大麻烦！因此，在 Go 中，作为 API 函数的作者，你一定**不要将 panic 当作错误返回给 API 调用者**。

到这里，我们已经基本讲完了函数健壮性设计要注意的各种事项，你一定要注意我前面提到的这几点。接下来，我们进入下一部分，看看函数的简洁性设计。

使用 defer 简化函数实现

对函数设计来说，如何实现简洁的目标是一个大话题。你可以从通用的设计原则去谈，比如函数要遵守单一职责，职责单一的函数肯定要比担负多种职责的函数更简单。你也可以从函数实现的规模去谈，比如函数体的规模要小，尽量控制在 80 行代码之内等。

但我们这个是 Go 语言的课程，所以我们的角度更侧重于 **Go 中是否有现成的语法元素，可以帮助我们简化 Go 函数的设计和实现**。我也把答案剧透给你，有的，它就是 **defer**。

同样地，我们也用一个具体的例子来理解一下。日常开发中，我们经常会编写一些类似下面示例中的伪代码：

复制代码

```
func doSomething() error {
    var mu sync.Mutex
    mu.Lock()

    r1, err := OpenResource1()
    if err != nil {
        mu.Unlock()
        return err
    }

    r2, err := OpenResource2()
    if err != nil {
        r1.Close()
        mu.Unlock()
        return err
    }

    r3, err := OpenResource3()
    if err != nil {
        r2.Close()
        r1.Close()
        mu.Unlock()
        return err
    }

    // 使用r1，r2, r3
    err = doWithResources()
    if err != nil {
        r3.Close()
        r2.Close()
        r1.Close()
        mu.Unlock()
        return err
    }

```

```
36      r3.Close()
37      r2.Close()
38      r1.Close()
39      mu.Unlock()
40      return nil
41  }
```

我们看到，这类代码的特点就是在函数中会申请一些资源，并在函数退出前释放或关闭这些资源，比如这里的互斥锁 mu 以及资源 r1~r3 就是这样。

函数的实现需要确保，无论函数的执行流是按预期顺利进行，还是出现错误，这些资源在函数退出时都要被及时、正确地释放。为此，我们需要尤为关注函数中的错误处理，在错误处理时不能遗漏对资源的释放。

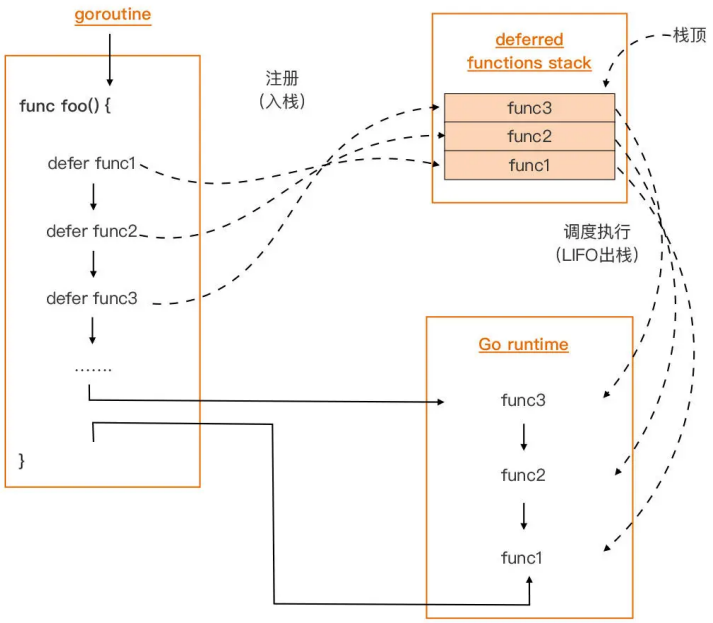
但这样的要求，就导致我们在进行资源释放，尤其是有多个资源需要释放的时候，比如上面示例那样，会大大增加开发人员的心智负担。同时当待释放的资源个数较多时，整个代码逻辑就会变得十分复杂，程序可读性、健壮性也会随之下降。但即便如此，如果函数实现中的某段代码逻辑抛出 panic，传统的错误处理机制依然没有办法捕获它并尝试从 panic 恢复。

Go 语言引入 defer 的初衷，就是解决这些问题。那么，defer 具体是怎么解决这些问题的呢？或者说，defer 具体的运作机制是怎样的呢？

defer 是 Go 语言提供的一种延迟调用机制，defer 的运作离不开函数。怎么理解呢？这句话至少有以下两点含义：

在 Go 中，只有在函数（和方法）内部才能使用 defer；

defer 关键字后面只能接函数（或方法），这些函数被称为 **deferred 函数**。defer 将它们注册到其所在 Goroutine 中，用于存放 deferred 函数的栈数据结构中，这些 deferred 函数将在执行 defer 的函数退出前，按后进先出（LIFO）的顺序被程序调度执行（如下图所示）。



极客时间

而且，无论是执行到函数体尾部返回，还是在某个错误处理分支显式 return，又或是出现 panic，已经存储到 deferred 函数栈中的函数，都会被调度执行。所以说，deferred 函数是一个可以在任何情况下为函数进行**收尾工作**的好“伙伴”。

我们回到刚才的那个例子，如果我们把收尾工作挪到 deferred 函数中，那么代码将变成如下这个样子：

复制代码

```
func doSomething() error {
    var mu sync.Mutex
    mu.Lock()
    defer mu.Unlock()

    r1, err := OpenResource1()
    if err != nil {
        return err
    }
    defer r1.Close()

    r2, err := OpenResource2()
    if err != nil {
        return err
    }
    defer r2.Close()

    r3, err := OpenResource3()
    if err != nil {
```

```
20          return err
21      }
22      defer r3.Close()
23
24      // 使用r1，r2, r3
25      return doWithResources()
26  }
```

我们看到，使用 defer 后对函数实现逻辑的简化是显而易见的。而且，这里资源释放函数的 defer 注册动作，紧邻着资源申请成功的动作，这样成对出现的惯例就极大降低了遗漏资源释放的可能性，我们开发人员也不用再小心翼翼地在每个错误处理分支中检查是否遗漏了某个资源的释放动作。同时，代码的简化也意味代码可读性的提高，以及代码健壮度的增强。

那我们日常开发中使用 defer，有没有什么要特别注意的呢？

defer 使用的几个注意事项

大多数 Gopher 都喜欢 defer，因为它不仅可以用来捕捉和恢复 panic，还能让函数变得更简洁和健壮。但“工欲善其事，必先利其器“，一旦你要用 defer，有几个关于 defer 使用的注意事项是你一定要提前了解清楚的，可以避免掉进一些不必要的“坑”。

第一点：明确哪些函数可以作为 deferred 函数

这里，你要清楚，对于自定义的函数或方法，defer 可以给与无条件的支持，但是对于有返回值的自定义函数或方法，返回值会在 deferred 函数被调度执行的时候被自动丢弃。

而且，Go 语言中除了自定义函数 / 方法，还有 Go 语言内置的 / 预定义的函数，这里我给出了 Go 语言内置函数的完全列表：

复制代码

```
Functions:
  append cap close complex copy delete imag len
  make new panic print println real recover
```

那么，Go 语言中的内置函数是否都能作为 deferred 函数呢？我们看下面的示例：

复制代码

```
// defer1.go

 func bar() (int, int) {
     return 1, 2
 }

 func foo() {
     var c chan int
     var sl []int
     var m = make(map[string]int, 10)
     m["item1"] = 1
     m["item2"] = 2
     var a = complex(1.0, -1.4)

     var sl1 []int
     defer bar()
     defer append(sl, 11)
     defer cap(sl)
     defer close(c)
     defer complex(2, -2)
     defer copy(sl1, sl)
     defer delete(m, "item2")
     defer imag(a)
     defer len(sl)
     defer make([]int, 10)
     defer new(*int)
     defer panic(1)
     defer print("hello, defer\n")
     defer println("hello, defer")
     defer real(a)
     defer recover()
 }

 func main() {
     foo()
 }
```

运行这个示例代码，我们可以得到：

复制代码

```
$go run defer1.go
# command-line-arguments
./defer1.go:17:2: defer discards result of append(sl, 11)
./defer1.go:18:2: defer discards result of cap(sl)
./defer1.go:20:2: defer discards result of complex(2, -2)
./defer1.go:23:2: defer discards result of imag(a)
./defer1.go:24:2: defer discards result of len(sl)
./defer1.go:25:2: defer discards result of make([]int, 10)
```

```
./defer1.go:26:2: defer discards result of new(*int)
./defer1.go:30:2: defer discards result of real(a)
```

我们看到，Go 编译器居然给出一组错误提示！

从这组错误提示中我们可以看到，append、cap、len、make、new、imag 等内置函数都是不能直接作为 deferred 函数的，而 close、copy、delete、print、recover 等内置函数则可以直接被 defer 设置为 deferred 函数。

不过，对于那些不能直接作为 deferred 函数的内置函数，我们可以使用一个包裹它的匿名函数来间接满足要求，以 append 为例是这样的：

复制代码

```
defer func() {
  _ = append(sl, 11)
}()
```

第二点：注意 defer 关键字后面表达式的求值时机

这里，你一定要牢记一点：**defer 关键字后面的表达式，是在将 deferred 函数注册到 deferred 函数栈的时候进行求值的。**

我们同样用一个典型的例子来说明一下 defer 后表达式的求值时机：

复制代码

```
func foo1() {
    for i := 0; i <= 3; i++ {
        defer fmt.Println(i)
    }
}

func foo2() {
    for i := 0; i <= 3; i++ {
        defer func(n int) {
            fmt.Println(n)
        }(i)
    }
}

func foo3() {

```

```
    for i := 0; i <= 3; i++ {
        defer func() {
            fmt.Println(i)
        }()
    }
}

func main() {
    fmt.Println("foo1 result:")
    foo1()
    fmt.Println("\nfoo2 result:")
    foo2()
    fmt.Println("\nfoo3 result:")
    foo3()
}
```

这里，我们一个个分析 foo1、foo2 和 foo3 中 defer 后的表达式的求值时机。

首先是 foo1。foo1 中 defer 后面直接用的是 fmt.Println 函数，每当 defer 将 fmt.Println 注册到 deferred 函数栈的时候，都会对 Println 后面的参数进行求值。根据上述代码逻辑，依次压入 deferred 函数栈的函数是：

复制代码

```
fmt.Println(0)
fmt.Println(1)
fmt.Println(2)
fmt.Println(3)
```

因此，当 foo1 返回后，deferred 函数被调度执行时，上述压入栈的 deferred 函数将以 LIFO 次序出栈执行，这时的输出的结果为：

复制代码

```
3
2
1
0
```

然后我们再看 foo2。foo2 中 defer 后面接的是一个带有一个参数的匿名函数。每当 defer 将匿名函数注册到 deferred 函数栈的时候，都会对该匿名函数的参数进行求值。根据上述代码逻辑，依次压入 deferred 函数栈的函数是：

复制代码

```
func(0)
func(1)
func(2)
func(3)
```

因此，当 foo2 返回后，deferred 函数被调度执行时，上述压入栈的 deferred 函数将以 LIFO 次序出栈执行，因此输出的结果为：

复制代码

```
3
2
1
0
```

最后我们来看 foo3。foo3 中 defer 后面接的是一个不带参数的匿名函数。根据上述代码逻辑，依次压入 deferred 函数栈的函数是：

复制代码

```
func()
func()
func()
func()
```

所以，当 foo3 返回后，deferred 函数被调度执行时，上述压入栈的 deferred 函数将以 LIFO 次序出栈执行。匿名函数会以闭包的方式访问外围函数的变量 i，并通过 Println 输出 i 的值，此时 i 的值为 4，因此 foo3 的输出结果为：

复制代码

```
4
4
4
4
```

通过这些例子，我们可以看到，无论以何种形式将函数注册到 defer 中，deferred 函数的参数值都是在注册的时候进行求值的。

第三点：知晓 defer 带来的性能损耗

通过前面的分析，我们可以看到，defer 让我们进行资源释放（如文件描述符、锁）的过程变得优雅很多，也不易出错。但在性能敏感的应用中，defer 带来的性能负担也是我们必须要知晓和权衡的问题。

这里，我们用一个性能基准测试（Benchmark），直观地看看 defer 究竟会带来多少性能损耗。基于 Go 工具链，我们可以很方便地为 Go 源码写一个性能基准测试，只需将代码放在以“_test.go”为后缀的源文件中，然后利用 testing 包提供的“框架”就可以了，我们看下面代码：

复制代码

```
// defer_test.go
package main

import "testing"

func sum(max int) int {
    total := 0
    for i := 0; i < max; i++ {
        total += i
    }

    return total
}

func fooWithDefer() {
    defer func() {
        sum(10)
    }()
}
func fooWithoutDefer() {
    sum(10)
}

func BenchmarkFooWithDefer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fooWithDefer()
    }
}
func BenchmarkFooWithoutDefer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fooWithoutDefer()
    }
}
```

这个基准测试包含了两个测试用例，分别是 BenchmarkFooWithDefer 和 BenchmarkFooWithoutDefer。前者测量的是带有 defer 的函数执行的性能，后者测量的是不带有 defer 的函数的执行的性能。

在 Go 1.13 前的版本中，defer 带来的开销还是很大的。我们先用 Go 1.12.7 版本来运行一下上述基准测试，我们会得到如下结果：

复制代码

```
$go test -bench . defer_test.go
goos: darwin
goarch: amd64
BenchmarkFooWithDefer-8         30000000          42.6 ns/op
BenchmarkFooWithoutDefer-8      300000000           5.44 ns/op
PASS
ok      command-line-arguments  3.511s
```

从这个基准测试结果中，我们可以清晰地看到：**使用 defer 的函数的执行时间是没有使用 defer 函数的 8 倍左右。**

但从 Go 1.13 版本开始，Go 核心团队对 defer 性能进行了多次优化，到现在的 Go 1.17 版本，defer 的开销已经足够小了。我们看看使用 Go 1.17 版本运行上述基准测试的结果：

复制代码

```
$go test -bench . defer_test.go
goos: darwin
goarch: amd64
BenchmarkFooWithDefer-8         194593353           6.183 ns/op
BenchmarkFooWithoutDefer-8      284272650           4.259 ns/op
PASS
ok      command-line-arguments  3.472s
```

我们看到，带有 defer 的函数执行开销，仅是不带有 defer 的函数的执行开销的 1.45 倍左右，已经达到了几乎可以忽略不计的程度，我们可以放心使用。

小结

好了，今天的课讲到这里就结束了。在这一讲中，我们延续上一节的脉络，讲解了函数设计过程中应该考虑的、健壮性与简洁性方面的内容。

在函数健壮性方面，我给出了“三不要”原则，这三个原则你一定要记住。这里我们重点讲解了第三个原则：**不要假定异常不会发生**。借此，我们认识了 Go 语言中表示异常的 panic，也学习了 panic 发生后的代码执行流程。基于 panic 的行为特征，我们给出了 Go 函数设计过程中应对 panic 的三点经验，这里你要注意，“评估程序对 panic 的忍受度”是我们选取应对 panic 措施的前提。

另外，对于来自像 Java 这样的、基于 Exception 进行错误处理的编程语言的 Go 初学者们，切记不要将 panic 与错误处理混淆。

接下来，我们又讲解了如何让函数实现更加简洁。简洁性对于函数来说意味着可读性更好，更易于理解，也有利于我们代码健壮性的提升。Go 语言层面提供的 defer 机制可用于简化函数实现，尤其是在函数申请和释放资源个数较多的情况下。

如果我们要用好 defer，前提就是要了解 defer 的运作机制，这里你要把握住两点：

函数返回前，deferred 函数是按照后入先出（LIFO）的顺序执行的；

defer 关键字是在注册函数时对函数的参数进行求值的。

最后，在最新 Go 版本 Go1.17 中，使用 defer 带来的开销几乎可以忽略不计了，你可以放心使用。

思考题

defer 是 Gopher 们都喜欢的语言机制，那么我想请你思考一下，除了捕捉 panic、延迟释放资源外，我们日常编码中还有哪些使用 defer 的小技巧呢？一个小提示：你可以阅读一下 Go 标准库中关于 defer 的使用方法，看看是否能总结出一些小 tips。

欢迎你把这节课分享给更多对 Go 语言函数感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

分享给需要的人，Ta阅后你可得 20 元现金奖励

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 22 | 函数：怎么结合多返回值进行错误处理？

下一篇 24 | 方法：理解"方法"的本质

更多学习推荐

精选留言 (10)

写留言

return
2021-12-06

老师讲的太好了，
再请教一下老师， panic应该在什么场景下使用，
感觉平时 业务代码 使用 err足够，没有使用panic的场景，
文中举例的 http server比较合适， 但是json encode，虽然官方用了panic，但是个人感觉使用err也没毛病， 传入的 json串 有问题嘛，就是 入参校验err呗。 ...

展开

作者回复: 如果在平时panic用的，那么就说明你走在了正确的道路上:)。panic本身就是异常，异常本身就是很少发生的。目前看panic最常使用的就是充当“断言”，提示潜在bug时用的最多。

共 2 条评论 > 4

进化菌
2021-12-06

panic 恐慌，慌慌张张的抛出异常；recover 恢复，淡定还有事情要做，先别退出；defer 延迟，晚点要做的事情不能忘……

展开

作者回复: 👍

2

liaomars
2021-12-06

看了官方标准库的defer说明，发现一个tips是：可以跟踪函数的执行过程

作者回复: 可以的。在后面的一个实战项目中，我们就会应用到这一点。

共 2 条评论 > 2

宋世杰
2021-12-06

Defer函数被放在一个不同于主函数的函数栈中吗？ 是用slice实现的吗？

作者回复: 是go runtime层实现的。

共 2 条评论 > 2

一步
2021-12-07

请教一个非课程的问题：
v := [...]int{1: 2, 3: 4}
fmt.Println(len(v))

这里为什么打印出来的 长度是 4呢？

展开

共 2 条评论 1

helloxiaomin

2021-12-07

defer的使用技巧：1）defer要在panic之前，才能执行；2）defer内部的recover只能捕获当前协程的Panic，不能跨协程执行；3）recover只能在defer中调用才能生效。暂时想到这些

1

罗杰

2021-12-06

关于 defer 我还要好好看看，一直都没怎么理解透彻

展开

作者回复: 嗯嗯，如有问题，随时交流。

1

aoe

2021-12-08

在 main 函数开始加上这样的代码，可以从 panic 中无限复活了

```
defer func() {
    if e := recover(); e != nil {
        fmt.Println("recover the panic:", e)
    }...
```

展开

lesserror

2021-12-07

感谢 Tony Bai 老师关于defer的深入讲解，不过我想问一下，在for中使用defer函数，编辑器会提示可能造成资源泄露，能大致说明一下这块儿的知识吗？

展开

Bynow

2021-12-07

我用的goland ，文中更新后的bar代码：我的编辑器终端输出是
call main
call foo
call bar
exit foo...
展开

共 4 条评论